Artificial Software Diversification for WebAssembly

JAVIER CABRERA-ARTEAGA

Doctoral Thesis
Supervised by
Benoit Baudry and Martin Monperrus
Stockholm, Sweden, 2023

TRITA-EECS-AVL-2020:4
ISBN 100-

KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science
Division of Software and Computer Systems
SE-10044 Stockholm
Sweden

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av Teknologie doktorexamen i elektroteknik i .

Tryck: Universitetsservice US AB

Abstract

[1]

Keywords: Lorem, Ipsum, Dolor, Sit, Amet

Sammanfattning

[1]

LIST OF PAPERS

1. ***Superoptimization of WebAssembly Bytecode***
   Javier Cabrera-Arteaga,Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus
   Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (Programming 2021), MoreVMs
   `https://doi.org/10.1145/3397537.3397567`

2. ***CROW: Code Diversification for WebAssembly***
   Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
   Network and Distributed System Security Symposium (NDSS 2021), MADWeb
   `https://doi.org/10.14722/madweb.2021.23004`

3. ***Multi-Variant Execution at the Edge***
   Javier Cabrera-Arteaga,Pierre Laperdrix, Martin Monperrus, Benoit Baudry
   Conference on Computer and Communications Security (CCS 2022), Moving Target Defense (MTD)
   `https://dl.acm.org/doi/abs/10.1145/3560828.3564007`

4. ***WebAssembly Diversification for Malware Evasion***
   Javier Cabrera-Arteaga,Tim Toady, Martin Monperrus, Benoit Baudry
   Computers & Security, Volume 131, 2023
   `https://www.sciencedirect.com/science/article/pii/S0167404823002067`

5. ***Wasm-mutate: Fast and Effective Binary Diversification for WebAssembly***
   Javier Cabrera-Arteaga, Nick Fitzgerald, Martin Monperrus, Benoit Baudry

6. ***Scalable Comparison of JavaScript V8 Bytecode Traces***
   Javier Cabrera-Arteaga,Martin Monperrus, Benoit Baudry
   11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (SPLASH 2019)
   `https://doi.org/10.1145/3358504.3361228`

ACKNOWLEDGEMENT

ACRONYMS

List of commonly used acronyms:

Wasm WebAssembly

Contents

Part I

Thesis

01 INTRODUCTION

TODO Recent papers first. Mention Workshops instead in conference. "Procedings of XXXX". Add the pages in the papers list.

1.1 Background

TODO Motivate with the open challenges.

1.2 Problem statement

TODO Problem statement **TODO** Set the requirements as R1, R2, then map each contribution to them.

1.3 Automatic Software diversification requirements

1. **1:** **TODO** Requirement 1

1.4 List of contributions

C1: Methodology contribution: We propose a methodology for generating software diversification for WebAssembly and the assessment of the generated diversity.

C2: Theoretical contribution: We propose theoretical foundation in order to improve Software Diversification for WebAssembly.

C3: Automatic diversity generation for WebAssembly: We generate WebAssembly program variants.

C4: Software Diversity for Defensive Purposes: We assess how generated WebAssembly program variants could be used for defensive purposes.

C5: Software Diversity for Offensives Purposes: We assess how generated WebAssembly program variants could be used for offensive purposes, yet improving security systems.

Contribution	Resarch papers				
	P1	P2	P3	P4	P5
C1	x	x		x	x
C2	x	x			
C3	x	x	x		
C4	x	x	x		
C5			x		
C6	x	x	x	x	x

Table 1.1: Mapping of the contributions to the research papers appended to this thesis.

C6: Software Artifacts: We provide software artifacts for the research community to reproduce our results.

TODO Make multi column table

1.5 Summary of research papers

P1: Superoptimization of WebAssembly Bytecode.

P2: CROW: Code randomization for WebAssembly bytecode.

P3: Multivariant execution at the Edge.

P4: Wasm-mutate: Fast and efficient software diversification for WebAssembly.

P5: WebAssembly Diversification for Malware evasion.

1.6 Thesis outline

02 Background and State of the Art

2.1 WebAssembly

TODO The concept of managed and unmanaged data in Wasm from Lehman. **TODO** Define: managed and unmanaged data, sections, execution stack, etc.

2.1.1 WebAssembly datatypes

2.1.2 WebAssembly memory model

TODO Managed and unmanaged

2.1.3 WebAssembly execution model

TODO Stack, frames and blocks **TODO** First order breaks

2.1.4 WebAssembly ecosystems

TODO Mention, stress the landscape of tools that involve Wasm. Include analysis tools, fuzzers, optimizers and malware detectors.
TODO End up motivating the need of Software Diversification for: testing and reliability.

2.2 Software diversification

2.3 Generating Software Diversification

2.3.1 Variants generation

2.3.2 Variants equivalence

TODO Automatic, SMT based **TODO** Take a look to Jackson thesis, we have a similar problem he faced with the superoptimization of NaCL

TODO By design **TODO** Introduce the notion of rewriting rule by Sasnaukas.

2.4 Exploiting Software Diversification

2.4.1 Defensive Diversification

2.4.2 Offensive Diversification

03 AUTOMATIC SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

WebAssembly programs are produced ahead of time through a process that begins with the source code of the program and moves through a compiler, resulting in a WebAssembly program. Software Diversification can be achieved at any of these stages. Diversifying at the source code stage, however, is not practical due to the need of creating a distinct diversifier for each language compatible with WebAssembly. In contrast, focusing on the compiler stage presents a viable option, especially considering that 70% of WebAssembly binaries are created using LLVM-based compilers, as noted by Hilbig et al. [?]. Furthermore, implementing diversification at the WebAssembly program stage stands as the most generic strategy, applicable to any WebAssembly program in the wild. Therefore, this thesis focuses to the exploration of diversification strategies at the compiler and WebAssembly program stages, employing two main approaches: compiler-based and binary-based.

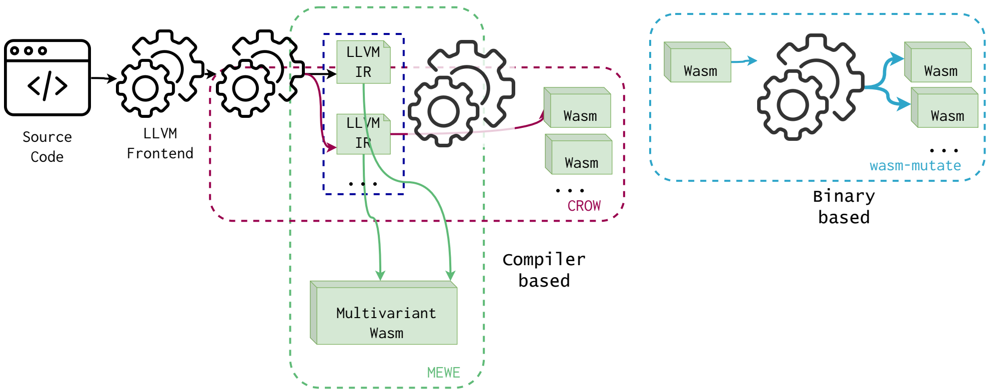

Figure 3.1: Approach landscape.

Our compiler-based strategies are depicted in red and green in Figure 3.1. This approach introduces a diversifier component in the LLVM pipeline, generating LLVM IR variants and producing artificial software diversity for Wasm. This strategy encompasses two tools: CROW [**?**], which creates Wasm program variants, and MEWE [**?**], which merges these variants to foster multivariant execution for Wasm. In contrast, the binary-based strategy, illustrated in blue in Figure 3.1, offers diversification for any WebAssembly program, removing the need for compiler tuning. WASM-MUTATE [**?**] generates a pool of WebAssembly program variants through rewriting rules upon an e-graph [**?**] data structure.

This dissertation contributes to the field of Software Diversification for WebAssembly, presenting three main technical contributions: CROW, MEWE, and WASM-MUTATE, which will be elaborated upon in the subsequent sections. Besides, we compare our three technical contributions between them.

3.1 CROW: Code Randomization of WebAssembly

This section details CROW [**?**], represented as the red squared tooling in Figure 3.1. CROW is designed to produce semantically equivalent Wasm variants from the output of an LLVM front-end, utilizing a custom Wasm LLVM backend.

Figure 3.2 illustrates CROW's workflow in generating program variants, a process compound of two core stages: *exploration* and *combining*. During the *exploration* stage, CROW processes every instruction within each function of the LLVM input, creating a set of functionally equivalent code variants. This process ensures a rich pool of options for the subsequent stage. In the *combining* stage, these alternatives are assembled to form diverse LLVM IR variants, a task achieved through the exhaustive traversal of the power set of all potential combinations of code replacements. The final step involves the custom Wasm LLVM backend, which compiles the crafted LLVM IR variants into Wasm binaries.

3.1.1 Variants' generation

The primary component of CROW's exploration process is its code replacements generation strategy. The diversifier implemented in CROW is based on the proposed superdiversifier methodology of Jacob et al. [**?**]. A superoptimizer focuses on *searching* for a new program that is faster or smaller than the original code while preserving its functionality. The concept of superoptimizing a program dates back to 1987, with the seminal work of Massalin [**?**] which proposes an exhaustive exploration of the solution space. The search space is defined by choosing a subset of the machine's instruction set and generating combinations of optimized programs, sorted

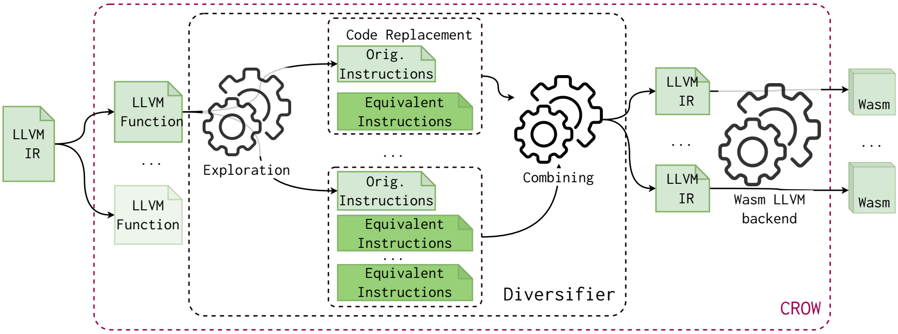

Figure 3.2: CROW components following the diagram in Figure 3.1. CROW takes LLVM IR to generate functionally equivalent code replacements. Then, CROW assembles program variants by combining them. Figure taken from [?].

by code size in ascending order. If any of these programs is found to perform the same function as the source program, the search halts. On the contrary, a superdiversifier keeps all intermediate search results despite their performance.

We modify Souper [?] to keep all possible solutions in their searching algorithm. Souper builds a Data Flow Graph for each LLVM integer-returning instruction. Then, for each Data Flow Graph, Souper exhaustively builds all possible expressions from a subset of the LLVM IR language. Each syntactically correct expression in the search space is semantically checked versus the original with a theorem solver. Souper synthesizes the replacements in increasing size. Thus, the first found equivalent transformation is the optimal replacement result of the searching. CROW keeps more equivalent replacements during the searching by removing the halting criteria. Instead the original halting conditions, CROW does not halt when it finds the first replacement. CROW continues the search until a timeout is reached or the replacements grow to a size larger that a predefined threshold.

Notice that the searching space increases exponentially with the size of the LLVM IR language subset. Thus, we prevent Souper from synthesizing instructions with no correspondence in the Wasm backend. This decision reduces the searching space. For example, creating an expression having the `freeze` LLVM instructions will increase the searching space for instruction without a Wasm's opcode in the end. Moreover, we disable the majority of the pruning strategies of Souper for the sake of more program variants. For example, Souper prevents the generation of the commutative operations during the searching. On the contrary, CROW still uses such transformation as a strategy to generate program variants.

The last stage involves the custom Wasm LLVM backend, which is in charge of generating the Wasm programs. For it, we have the premise of removing all built-in optimizations in the LLVM backend that could reverse Wasm variants. We disable all optimizations in the Wasm backend that could reverse the CROW transformations.

3.1.2 Constant inferring

CROW, through using Souper adds a new transformation strategy, *constant inferring*. This means that Souper infers pieces of code as a single constant assignment. In particular, Souper focuses on variables that are used to control branches. After a *constant inferring*, the generated program is considerably different from the original program, being suitable for diversification.

Let us illustrate the case with an example. The Babbage problem code in Listing 3.1 is composed of a loop that stops when it discovers the smaller number that fits with the Babbage condition in Line 4.

Listing 3.1: Babbage problem.

```
1   int babbage() {
2       int current = 0,
3           square;
4       while ((square=current*current) %
            ↪ 1000000 != 269696) {
5           current++;
6       }
7       printf ("The number is %d\n",
            ↪ current);
8       return 0 ;
9   }
```

Listing 3.2: Constant inferring transformation over the original Babbage problem in Listing 3.1.

```
int babbage() {
    int current = 25264;



    printf ("The number is %d\n", current
        ↪ );
    return 0 ;
}
```

In theory, this value can also be inferred by unrolling the loop the correct number of times with the LLVM toolchain. However, standard LLVM tools cannot unroll the **`while`**-loop because the loop count is too large. The original Souper deals with this case, generating the program in Listing 3.2. It infers the value of `current` in Line 2 such that the Babbage condition is reached. Therefore, the condition in the loop will always be false. Then, the loop is dead code and is removed in the final compilation. The new program in Listing 3.2 is remarkably smaller and faster than the original code. Therefore, it offers differences both statically and at runtime[1]

[1]Notice that for the sake of illustration, we show both codes in C language, this process inside CROW is performed directly in LLVM IR.

■ 3.1.3 Combining replacements

When we retarget Souper, to create variants, we recombine all code replacements, including those for which a constant inferring was performed. This allows us to create variants that are also better than the original program in terms of size and performance. Most of the Artificial Software Diversification works generate variants that are as performant or iller than the original program. By using a superdiversifier, we could be able to generate variants that are better, in terms of performance, than the original program. This will give the option to developers to decide between performance and diversification without sacrificing the former.

On the other hand, when Souper finds a replacement, it is applied to all equal instructions in the original LLVM binary. In our implementation, we apply the transformations one by one. For example, if we find a replacement that is suitable for N difference places in the original program, we generate N different programs by applying the transformation in only one place at a time. Notice that this strategy provides a combinatorial explosion of program variants as soon as the number of replacements increases.

■ 3.1.4 CROW instantiation

Let us illustrate how CROW works with the example code in Listing 3.3. The `f` function calculates the value of $2 * x + x$ where `x` is the input for the function. CROW compiles this source code and generates the intermediate LLVM bitcode in the left most part of Listing 3.4. CROW potentially finds two integer returning instructions to look for variants, as the right-most part of Listing 3.4 shows.

Listing 3.3: C function that calculates the quantity $2x + x$

```
int f(int x) {
    return 2 * x + x;
}
```

CROW, detects `code_1` and `code_2` as the enclosing boxes in the left most part of Listing 3.4 shows. CROW synthesizes $2 + 1$ candidate code replacements for each code respectively as the green highlighted lines show in the right most parts of Listing 3.4. The baseline strategy of CROW is to generate variants out of all possible combinations of the candidate code replacements, *i.e.,* uses the power set of all candidate code replacements.

In the example, the power set is the cartesian product of the found candidate code replacements for each code block, including the original ones, as Listing 3.5 shows. The power set size results in 6 potential function variants. Yet, the generation stage would eventually generate 4 variants from the original program. CROW generated 4 statically different Wasm

Listing 3.4: LLVM's intermediate representation program, its extracted instructions and replacement candidates. Gray highlighted lines represent original code, green for code replacements.

```
define i32 @f(i32) {          Replacement candidates for Replacement candidates for
                                   code_1                      code_2
2code 211.5103.5cm
1code 111.53.53.0cm           %2 = mul nsw i32 %0,2       %3 = add nsw i32 %0,%2
%2 = mul nsw i32 %0,2
%3 = add nsw i32 %0,%2        %2 = add nsw i32 %0,%0      %3 = mul nsw %0, 3:i32

ret i32 %3                    %2 = shl nsw i32 %0, 1:i32
}

define i32 @main() {
%1 = tail call i32 @f(
     i32 10)
ret i32 %1
}
```

Listing 3.5: Candidate code replacements combination. Orange highlighted code illustrate replacement candidate overlapping.

```
%2 = mul nsw i32 %0,2                        %2 = mul nsw i32 %0,2
%3 = add nsw i32 %0,%2                       %3 = mul nsw %0, 3:i32

%2 = add nsw i32 %0,%0                       %2 = add nsw i32 %0,%0
%3 = add nsw i32 %0,%2                       %3 = mul nsw %0, 3:i32

%2 = shl nsw i32 %0, 1:i32                   %2 = shl nsw i32 %0, 1:i32
%3 = add nsw i32 %0,%2                       %3 = mul nsw %0, 3:i32
```

files, as Listing 3.6 illustrates. This gap between the potential and the actual number of variants is a consequence of the redundancy among the bitcode variants when composed into one. In other words, if the replaced code removes other code blocks, all possible combinations having it will be in the end the same program. In the example case, replacing `code_2` by `mul nsw %0, 3`, turns `code_1` into dead code, thus, later replacements generate the same program variants. The rightmost part of Listing 3.5 illustrates how for three different combinations, CROW produces the same variant. We call this phenomenon a *code replacement overlapping*.

One might think that a reasonable heuristic could be implemented to avoid such overlapping cases. Instead, we have found it easier and faster to generate the variants with the combination of the replacement and check their uniqueness after the program variant is compiled. This prevents us from having an expensive checking for overlapping inside the CROW code. Still, this phenomenon calls for later optimizations in future works.

Listing 3.6: Wasm program variants generated from program Listing 3.3.

```
func $f (param i32) (result i32)
  local.get 0
    i32.const 2
    i32.mul
    local.get 0
    i32.add
```

```
func $f (param i32) (result i32)
    local.get 0
    i32.const 1
    i32.shl
    local.get 0
    i32.add
```

```
func $f (param i32) (result i32)
    local.get 0
    local.get 0
    i32.add
    local.get 0
    i32.add
```

```
func $f (param i32) (result i32)
    local.get 0
    i32.const 3
    i32.mul
```

3.2 MEWE: Multi-variant Execution for WebAssembly

This section describes MEWE [?]. MEWE synthesizes diversified function variants by using CROW. It then provides execution-path randomization in a Multivariant Execution (MVE). The tool generates application-level multivariant binaries without changing the operating system or Wasm runtime. MEWE creates an MVE by intermixing functions for which CROW generates variants, as the green squared tooling in Figure 3.1 shows. MEWE inlines function variants when appropriate, resulting in call stack diversification at runtime.

In Figure 3.3 we zoom MEWE from the blue highlighted square in Figure 3.1. MEWE takes the LLVM IR variants generated by CROW's diversifier. It then merges LLVM IR variants into a Wasm multivariant. In the figure, we highlight the two components of MEWE, *Multivariant Generation* and the *Mixer*. In the *Multivariant Generation* process, MEWE merges the LLVM IR variants created by CROW and creates an LLVM multivariant binary. The merging of the variants intermixes the calling of function variants, allowing the execution path randomization.

The Mixer augments the LLVM multivariant binary with a random generator. The random generator is needed to perform the execution-path randomization. Also, *The Mixer* fixes the entrypoint in the multivariant binary. Finally, MEWE generates a standalone multivariant Wasm binary using the same custom Wasm LLVM backend from CROW. Once generated, the multivariant Wasm binary can be deployed to any Wasm engine.

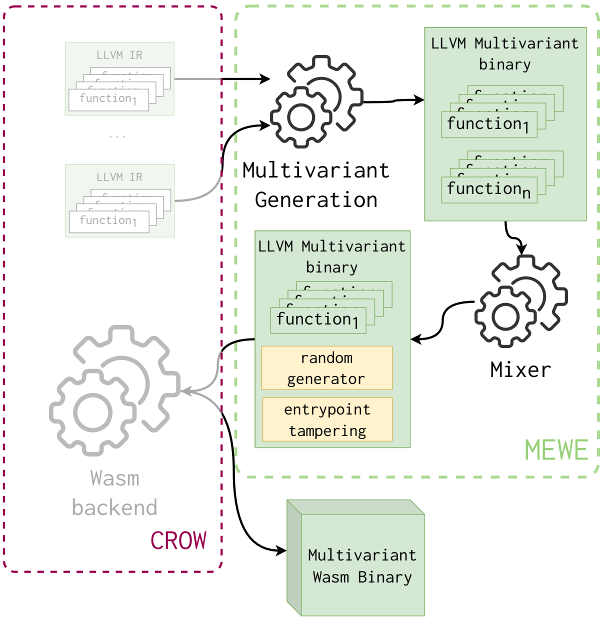

Figure 3.3: Overview of MEWE workflow. It takes as input an LLVM binary. It first generates a set of functionally equivalent variants for each function in the binary using CROW. Then, MEWE generates an LLVM multivariant binary composed of all the function variants. Finally, the Mixer includes the behavior in charge of selecting a variant when a function is invoked. Finally, the MEWE mixer composes the LLVM multivariant binary with a random number generation library and tampers the original application entrypoint. The final process produces a Wasm multivariant binary ready to be deployed. Figure taken from [**?**].

3.2.1 Multivariant generation

The key component of MEWE consists in combining the variants into a single binary. The goal is to support execution-path randomization at runtime. The core idea is to introduce one dispatcher function per original function with variants. A dispatcher function is a synthetic function in charge of choosing a variant at random when the original function is called. With the introduction of the dispatcher function, MEWE turns the original call graph into a multivariant call graph, defined as follows.

In Figure 3.4, we show the original static call graph for an original program (top of the figure), as well as the multivariant call graph generated with MEWE (bottom of the figure). The gray nodes represent function variants, the green nodes function dispatchers, and the yellow nodes are the original functions. The directed edges represent the possible calls. The

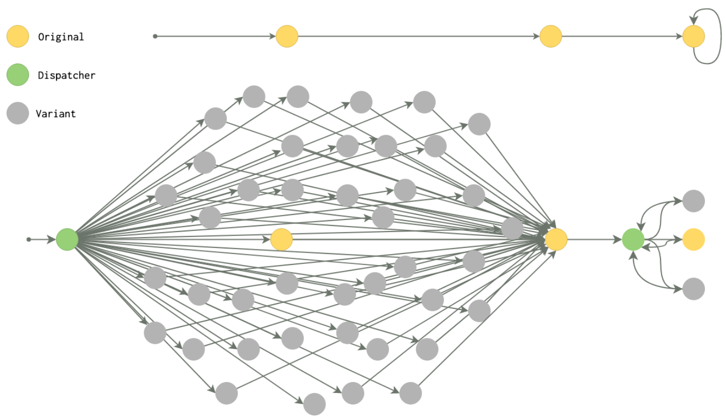

Figure 3.4: Example of two static call graphs. At the top, the original call graph, at the bottom, the multivariant call graph, which includes nodes that represent function variants (in gray), dispatchers (in green), and original functions (in yellow). Figure taken from [**?**].

original program includes three functions. MEWE generates 43 variants for the first function, none for the second, and three for the third. MEWE introduces two dispatcher nodes for the first and third functions. Each dispatcher is connected to the corresponding function variants to invoke one variant randomly at runtime.

In Listing 3.7, we illustrate the LLVM construction for the function dispatcher corresponding to the right most green node of Figure 3.4. It first calls the random generator, which returns a value used to invoke a specific function variant. We implement the dispatchers with a switch-case structure to avoid indirect calls that can be susceptible to speculative execution-based attacks [**?**]. The choice of a switch-case also avoids having multiple function definitions with the same signature, which could increase the attack surface in case the function signature is vulnerable [**?**]. This also allows MEWE to inline function variants inside the dispatcher instead of defining them again.

■ 3.2.2 MEWE's Mixer

MEWE has four specific objectives: link the LLVM multivariant binary, inject a random generator, tamper the application's entrypoint, and merge all these components into a multivariant Wasm binary. We use the Rustc compiler[2] to orchestrate the mixing. For the random generator, we rely on WASI's specification [**?**] for the random behavior of the dispatchers. However, its exact implementation is dependent on the platform on which the binary is deployed. The Mixer creates a new entrypoint for the binary

[2] `https://doc.rust-lang.org/rustc/what-is-rustc.html`

```
define internal i32 @foo(i32 %0) {
  entry:
    %1 = call i32 @discriminate(i32 3)
    switch i32 %1, label %end [
      i32 0, label %case_43_
      i32 1, label %case_44_
    ]
  case_43_:
    %2 = call i32 @foo_43_(%0)
    ret i32 %2
  case_44_:
    %3 = <body of foo_44_ inlined>
    ret i32 %3
  end:
    %4 = call i32 @foo_original(%0)
    ret i32 %4
}
```

Listing 3.7: Dispatcher function embedded in the multivariant binary of the original function in the rightmost green node in Figure 3.4.

called *entrypoint tampering*. It wraps the dispatcher for the entrypoint variants as a new function for the final Wasm binary and is declared as the application entrypoint.

■ 3.3 WASM-MUTATE: Fast and Effective Binary for WebAssembly

In this section we present our third technical contribution, WASM-MUTATE [**?**]. It takes a WebAssembly program as input and produces functionally equivalent variants. WASM-MUTATE is highlighted as the blue squared tooling in Figure 3.1, its central approach involves synthesizing program variants by substituting parts of the original binary using rewriting rules, boosted by diversification space traversals using e-graphs [**?**].

Figure 3.5 illustrates the workflow of WASM-MUTATE, which initiates with a WebAssembly binary as its input. The first step involves parsing this binary to create suitable abstractions, e.g. an intermediate representation. Subsequently, WASM-MUTATE utilizes predefined rewriting rules to construct an e-graph for the initial program, encapsulating all potential equivalent codes derived from the rewriting rules. Then, pieces of the original program are randomly substituted by the result of random e-graph traversals, resulting in a variant that maintains functional equivalence to the original binary. This assurance of semantic preservation is rooted in the inherent properties of the individual rewrite rules employed.

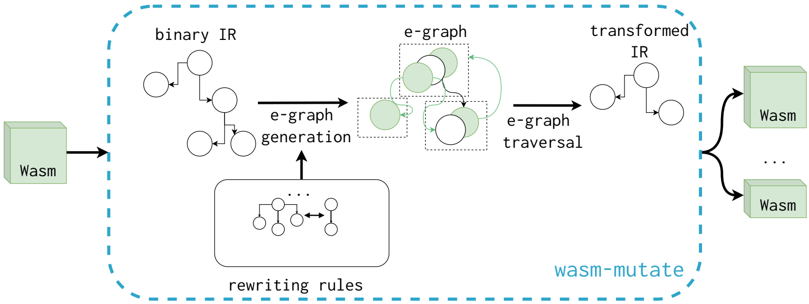

Figure 3.5: WASM-MUTATE high level architecture. It generates semantically equivalent variants from a given WebAssembly binary input. Its central approach involves synthesizing these variants by substituting parts of the original binary using rewriting rules, boosted by a diversification space traversals using e-graphs.

■ 3.3.1 WebAssembly Rewriting Rules

WASM-MUTATE incorporates a total of 135 rewriting rules, organized into categories referred to as meta-rules. The rewriting rules are conceived based on the seminal work of Sasnauskas et al. [?], extended to include a predicate to enforce the conditions for replacement. Each rule is articulated as a tuple, represented as `(LHS, RHS, Cond)`, where: `LHS` identifies the segment of code targeted for replacement, `RHS` outlines the functionally equivalent substitute, and `Cond` defines the circumstances permitting the replacements. For example, in the case of WebAssembly binaries, the `Cond` predicate ensures that the replacement does not violate the type constraints. The following text details seven prominent meta-rules utilized in WASM-MUTATE.

Add type: WASM-MUTATE implements two rewrite rules for the Type Section of input WebAssembly programs, one of which is illustrated in the following rewriting rule.

```
LHS (module
  (type (;0;) (func (param i32) (result i64)))
```

```
RHS (module
  (type (;0;) (func (param i32) (result i64)))
+ (type (;0;) (func (param i64) (result i32 i64)))
```

This transformation creates random function signatures, varying both, the number of parameters and the count of results. It ensures the consistency of the index of already defined types, even after the introduction of a new type.

Add function: WASM-MUTATE adds new random functions by mutating the code, section and function sections. This process begins with the creation of a random type signature, followed by the formulation of a random function body that simply returns the default value corresponding to the result type. An illustration of this rewriting rule is provided in the subsequent example.

```
LHS (module
      (type (;0;) (func (param i32 f32) (result i64)))
```

```
RHS (module
      (type (;0;) (func (param i32 f32) (result i64)))
   +_______(func (;0;) (type 0) (param i32 f32) (result i64)
   +__________i64.const 0)
```

Remove dead code: WASM-MUTATE randomly eliminates dead code, targeting specific elements such as *functions, types, custom sections, imports, tables, memories, globals, data segments, and elements* that are verifiably unused. For instance, the removal of a memory declaration needs the absence of any memory access operations within the binary code. WASM-MUTATE incorporates distinct mutators for each element type to facilitate this process. The following example showcases a function removal using this meta-rule.

```
LHS (module (type (func)))
```

```
RHS - (module (import "" "" (func)))

Cond The removed function is not called, it is not exported, and it is
     not in the binary _table.
```

Edit custom sections: WebAssembly's custom sections serve to save metadata, including details such as the compiler name responsible for generating the binary and symbol information pertinent to debugging. WASM-MUTATE alters custom sections. Through the *Edit Custom Section* transformation, it can randomly change either the content or the name of the custom section, a process illustrated in the subsequent rewriting rule example.

```
LHS (module
...
    - (@custom "CS42" "zzz..."
```

```
RHS (module
...
    + (@custom "RTY42" "xxx...")
```

If swapping: In WebAssembly, the if-construction is compound of two paths: the consequence and the alternative. The determination of which pathway to follow depends on the branching condition evaluated just before the `if` instruction. Specifically, a value greater than `0` at the top of the execution stack triggers the execution of the consequence code, while any other outcome initiates the alternative code. The *if swapping* rewriting rule interchanges the consequence and alternative codes within the if-construction, effectively reversing the original paths defined by the condition.

To facilitate the swapping of an if-construction in WebAssembly, WASM-MUTATE introduces a negation of the value situated at the top of the stack immediately preceding the `if` instruction. The methodology behind this rewriting is demonstrated in the following rewriting rule example.

```
LHS (module
    (func ...) (
 condition C
        (if A else B end)
    )
)
```

```
RHS (module
    (func ...) (
 condition C
i32.eqz
        (if B else A end)
    )
)
```

In this context, the consequence and alternative codes are labeled with the letters `A` and `B`, respectively, while the if-construction's condition is represented as `C`. To negate this condition, the `i32.eqz` instruction is incorporated into the RHS segment of the rewriting rule, functioning to compare the stack's top value with zero and, if true, pushing the value `1` onto the stack. In addition, WASM-MUTATE introduces a `nop` instruction to substitute for the absent code block, ensuring a seamless rewriting process.

Loop Unrolling: WASM-MUTATE randomly unrolls loops. Upon selecting a loop for unrolling, WASM-MUTATE segments its instructions using first-order breaks, i.e., jumps that lead back to the loop's beginning (See section ??). This strategy maintains the integrity of branching instructions overseeing the loop body, accounting for the need for label index adjustments during the unrolling phase. This principle applies equally to instructions guiding the transition to subsequent loop iterations.

As unrolling progresses, a fresh Wasm block emerges to contain both the replicated loop body and its original counterpart. Within this block, the divided groups of instructions find a new block, mirroring their original state but with a pivotal alteration: branching instructions exiting the loop body have their jump indices incremented by one. This adjustment is needed by the newly introduced `block ... end` scope wrapping the loop body, influencing the scope levels of the branching directives. The following example illustrates the unrolling process for a loop containing a single first-order break.

```
LHS (module
    (func ...) (
        (loop   A br_if 0   B end)
  )
)
```

```
RHS (module
    (func ...) (
        (block
            (block A'  br_if 0   B'  br 1   end)
            (loop   A'  br_if 0   B'  end)
        end)
    )
)
```

In the LHS part of the rewriting rule, the loop showcases a solitary first-order break, signaling that its activation prompts continuous loop iterations. The loop's operational span concludes just before the `end` instruction, delineating the juncture where the original loop ceases and the broader program execution resumes. When this loop is chosen for unrolling, it undergoes a bifurcation of its instructions into two distinct groups, termed `A` and `B`. The RHS part of the illustration brings to light the creation of two fresh Wasm blocks during the unrolling procedure. Here, the outer block is a repository for both the original and the duplicated loop bodies, while the inner entities, labeled `A'` and `B'`, embody alterations to the jump directives originally found in groups `A` and `B`.

Any jump directives within `A'` and `B'` that initially direct flow outside the loop need an increment of one in their jump indices. This modification is a response to the newly instituted block scope encircling the loop body in the course of unrolling. Moreover, the conclusion of the unrolled loop iteration's body is marked by the insertion of an unconditional branch. This strategic placement guarantees that, in the absence of a continuation in the loop body, the operation exits the scope.

Peephole: This meta-rule focuses on the rewriting of instruction sequences found within function bodies, representing the lowest level of rewriting. In WASM-MUTATE, we have devised 125 rewriting rules specifically for this category. WASM-MUTATE is structured to ensure the determinism of the instructions selected for replacement. Therefore, any rewriting rule inside the Peephole meta-rule avoids instructions that might induce undefined behavior, e.g., function calls. Consequently, the scope of this meta-rule is confined to modifications in stack and memory operations, preserving the integrity of the control frame labels.

The peephole category rewriting rules are meticulously designed and manually verified. An instance of a rewriting rule in this category can be appreciated below:

```
LHS (x)
```

```
RHS (x i32.or x)

Cond x should be i32 type
```

The previous rewriting rule example implies that the `LHS` 'x' is to be replaced by an idempotent bitwise `i32.or` operation with itself, as soon as x, which can be any subexpression, leaves a value of type i32 in the exeution stack.

3.3.2 Extending peephole meta-rules with custom operators

As illustrated in Figure 3.5, the initial step in the process involves parsing an input WebAssembly program, generating an intermediate representation. This step facilitates the transition of the WebAssembly program to the next stages of WASM-MUTATE. This representation extends the textual Wat format. We augment it with custom operator instructions to enhance the transformation capabilities of WASM-MUTATE [**?**]. Custom operator instructions form part of the lowest level of transformation we provide in WASM-MUTATE, the Peephole meta-rule.

These custom operator instructions are designed to bolster WASM-MUTATE by utilizing well-established code diversification techniques. For example, we add a `container` operator instruction, which acts as a holder for multiple instructions, enabling transformations without altering the program's semantics. Below, we demonstrate a rewriting rule that leverages it to insert `nop` instructions into the any WebAssembly program place:

```
LHS x
```

```
RHS (container (x nop))
```

The instruction `x`, can potentially be substituted with `container (x nop)`. When converting back to the WebAssembly binary format from the intermediate representation, custom instructions are meticulously handled to retain the original functionality of the WebAssembly program. Concretely, the `container` custom operator is removed, the `nop` instruction is replaced with the `nop` opcode, and `x`, which denotes a single instruction, is encoded with the appropriate opcode.

The `container` is one of four custom operator instructions introduced in the WASM-MUTATE intermediate representation. Specifically, we have added: UseGlobals: this operator substitutes its operand with the setting and retrieval actions involving a newly created global variable. Constant unfolding: this operator, working with a constant numeric operand, statically generates two numbers whose sum equals the constant, followed by the addition of operations for these numbers. Rand: this operator injects random constant numbers into the program. Significantly, this adaptability enables WASM-MUTATE to embrace a wide array of code diversification techniques at instruction level.

■ 3.3.3 E-graphs

We developed WASM-MUTATE leveraging e-graphs, a specific graph data structure for representing rewriting rules [**?**]. Within an e-graph, there exist two distinct node types: e-nodes and e-classes. The former encapsulates either an operator or an operand present in the rewriting rule, while the latter groups e-nodes into equivalence classes, essentially serving as a composite virtual node that contains a collection of e-nodes. Consequently, each e-class contains at least one e-node, with edges delineating the operator-operand equivalence relations between e-nodes and e-classes.

In the context of WASM-MUTATE, the e-graph is constructed from a WebAssembly program, drawing upon the analysis of its expressions and operations as depicted in data flow graphs. This entails the transformation of each distinct expression, operator, and operand into e-nodes. Next steps involve identifying equivalent expressions based on the corpus of rewriting rules, thereby clustering analogous e-nodes into e-classes. This phase might also witness the introduction of new operators to the graph as e-nodes.

As the final step, e-nodes in e-classes are interlinked through edges to illustrate their equivalence relationships, a dynamic process that evolves with the integration of new rewriting rules until reaching a point of equality saturation [**?**].

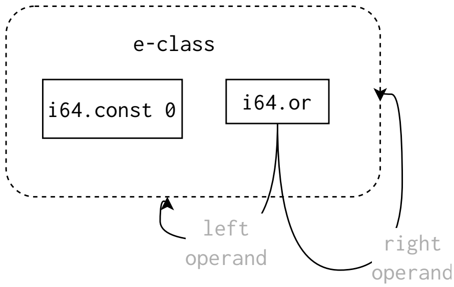

Figure 3.6: e-graph for idempotent bitwise-or rewriting rule. Solid lines represent operand-operator relations, and dashed lines represent equivalent class inclusion.

Let us illustrate the e-graph construction with a program that consists of a single instruction which returns an integer constant, denoted as `i64.const 0`. Besides, assume a single rewriting rule defined as `(x, x i64.or x, x returns type i64)`. In this scenario, the control flow graph of the program is quite straightforward, containing only a single node that stands for the lone instruction present. This rewriting rule is designed to express the equivalence of conducting an `or` operation where both operands are identical.

In Figure 3.6 we visualize how the e-graph data structure out of the program and rewriting rule. The process begins with the incorporation of the solitary program instruction, `i64.const 0`, as an e-node, which is represented by the leftmost solid rectangle node in the diagram. Then, we derive additional e-nodes from the rewriting rule, which is depicted by the rightmost solid rectangle in the figure. This involves introducing a fresh e-node labeled `i64.or` and establishing edges leading to the `x` e-node. Next, the equivalence by merging the two e-nodes is affirmed, thus forming a unified e-class. This step needs the redirection of the existing edges to target the newly formed e-class associated with the `x` symbol. Finally, we successfully construct an e-graph that encapsulates the relationships and equivalences dictated by the initial program and the rewriting rule, setting the stage for further analyses and transformations based on this structured representation.

■ 3.3.4 Random e-graph traversal for variants generation

Willsey et al. demonstrated the potential for high flexibility in extracting code fragments from e-graphs, a process that can be recursively orchestrated through a cost function applied to e-nodes and their respective operands. This methodology ensures the semantic equivalence of the derived code [**?**]. For instance, to extract the "optimal" code from an e-graph, one might commence the extraction at a specific e-node, subsequently selecting the AST with the minimal size from the available options within the corresponding e-class's operands. In omitting the cost function from the extraction strategy leads us to a significant property: *any path navigated through the e-graph yields a semantically equivalent code variant.*

The previous mentioned property is exploited in Figure 3.6, showcasing the feasibility of crafting an endless series of "or" operations. We exploit such property to generate diverse WebAssembly variants. We propose and implement an algorithm that facilitates the random traversal of an e-graph to yield semantically equivalent program variants, as detailed in Algorithm 1. This algorithm operates by taking an e-graph, an e-class node (starting with the root's e-class), and a parameter specifying the maximum extraction depth of the expression, to prevent infinite recursion. Within the algorithm, a random e-node is chosen from the e-class (as seen in lines 5 and 6), setting the stage for a recursive continuation with the offspring of the selected e-node (refer to line 8). Once the depth parameter reaches zero, the algorithm extracts the most concise expression available within the current e-class (line 3). Following this, the subexpressions are built (line 10) for each child node, culminating in the return of the complete expression (line 11).

Algorithm 1 e-graph traversal algorithm taken from [**?**].

```
1:  procedure TRAVERSE(egraph, eclass, depth)
2:      if depth = 0 then
3:          return smallest_tree_from(egraph, eclass)
4:      else
5:          nodes ← egraph[eclass]
6:          node ← random_choice(nodes)
7:          expr ← (node, operands = [])
8:          for each child ∈ node.children do
9:              subexpr ← TRAVERSE(egraph, child, depth − 1)
10:             expr.operands ← expr.operands ∪ {subexpr}
11:         return expr
```

■ 3.3.5 WASM-MUTATE instantiation

Let us illustrate how generates variant programs by using the before enunciated algorithm. Here, we use Algorithm 1 with maximum depth of 1. In Listing 3.8 a hypothetical original Wasm binary is illustrated. In this context, a potential user has set two pivotal rewriting rules: `(x, container (x nop),)` and `(x, x i32.add 0, x instanceof i32)`. The initial rule, which has been previously elaborated upon in 3.3, grants the ability to append a `nop` instruction to any subexpression within the program.

Conversely, the latter rule articulates the equivalence of augmenting any numeric value by zero. Both rules preserve the semantic of the original program when applied.

Leveraging the code presented in Listing 3.8 alongside the defined rewriting rules, we build the e-graph. The resulting e-graph can be found in Figure 3.7. This graphical representation illustrates the relationships between operators and operands, delineated through arrows connecting the respective nodes. The e-graph illustration highlights the potential pathways for rewriting according to the established rules.

```
(module
    (type (;0;) (func (param i32 f32) (result i64)))
    (func (;0;) (type 0) (param i32 f32) (result i64)
        i64.const 1)
)
```

Listing 3.8: Wasm function.

```
(module
    (type (;0;) (func (param i32 f32) (result i64)))
    (func (;0;) (type 0) (param i32 f32) (result i64)
        (i64.add (
            i64.const 0
            i64.const 1
            nop
        ))
    )
```

Listing 3.9: Random peephole mutation using egraph traversal for Listing 3.8 over e-graph Figure 3.7. The textual format is folded for better understanding.

In Figure 3.7, we highlight various stages of Algorithm 1 in the context of the scenario previously described. The algorithm initiates its journey at the e-class with the singular instruction `i64.const 1`, as seen in Listing 3.8. At (2), it randomly selects an equivalent node within the e-class, in this instance selecting the `i64.add` node, resulting: `expr = i64.add l r`.

As the traversal advances to (3), it focuses on the left operand of the previously chosen node, settling on the `i64.const 0` node within the same e-class. The process continues in (4), where the right operand of the `i64.add` node is choosen, selecting the `container` operator yielding: `expr = i64.or (i64.const 0 container ( r nop ))`.

At (5), the algorithm chooses the right operand of the `container`, which correlates to the initial instruction e-node highlighted in (6), culminating in the final expression: `expr = i64.or (i64.const 0 container( i64.const 1 nop)) i64.const 1`.

As we proceed to the subsequent encoding phases, the `container` operator is ignored as a real Wasm instruction, finally resulting the program in Listing 3.9.

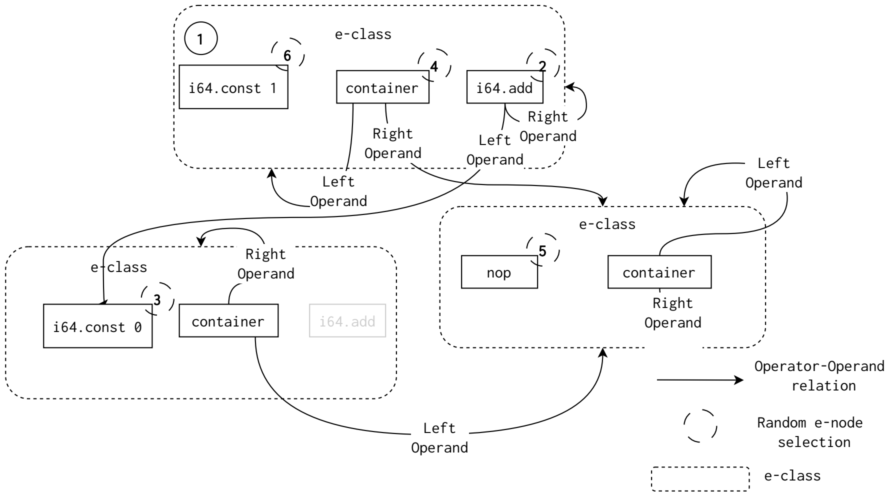

Figure 3.7: e-graph built for rewriting the first instruction of Listing 3.8.

Notice that, within the e-graph showcased in Figure 3.7, the container node maintains a presence across all e-classes. Consequently, increasing the depth parameter in Algorithm 1 would potentially escalate the number of viable variants infinitely.

3.4 Comparing CROW, MEWE and WASM-MUTATE

In this section we discuss the main differences between CROW, MEWE and wasm-mutate. We discuss the main differences between CROW, MEWE and wasm-mutate according to three main dimensions: 1) the technology and approach of each one, 2) the strength of the generated variants and, 3) the security guarantees of the variants generated by each tool. We select these three dimensions because they lead the implementation of our tools. Besides, we provide insights on the provided artifacts for each one of the tools and main implementation details.

3.4.1 Technology and approach

CROW is a compiler-based strategy, needing access to the original source code or its LLVM IR representation to work. Its core is a Satisfiability Modulo Theories (SMT) solver, ensuring the functional equivalence of the generated variants. This approach lays the groundwork for a universal LLVM superdiversifier, promising a wide range of applications and adaptability. Building upon CROW, MEWE extends its capabilities, utilizing the same underlying technology to create program variants. It goes a step further by packaging the LLVM IR variants into a Wasm multivariant.

On the other hand, WASM-MUTATE is a semi-automated, binary-based tool, centralizing its core around e-graph traversals. This approach facilitates the creation of a pool of WebAssembly program variants through the meticulous application of rewriting rules on an e-graph data structure. This method removes the need for compiler adjustments, offering compatibility with any existing WebAssembly binary and showcasing flexibility and applicability. Moreover, it introduces a versatile intermediate representation, establishing a general framework for binary rewriting in WebAssembly.

■ 3.4.2 Strength of the generated variants

CROW and MEWE use enumerative synthesis and verify semantic equivalence through SMT solvers. This approach not only has the potential to exceed handcrafted optimizations but also ensures that the transformations are preserved. In other words, the transformations generated out of CROW and MEWE are virtually irreversible, even following compiler optimizations. This is particularly remarkable in the case of *constant inferring* transformations. Furthermore, the use of enumerative synthesis allows the generated variants to potentially improve the original program's runtime performance, demystifying that software diversification inherently compromises performance. While CROW and MEWE do not require any extra input but the program to diversify, the speed of variant generation is intrinsically linked to the SMT solvers' efficiency, known to be slow. Besides, their variants generation capabilities are limited by the *overlapping* phenomenon discussed in 3.1.

On the other hand, WASM-MUTATE adopts a semi-automatic approach, requiring users to set the rewriting rules. Thus, the responsibility of ensuring functional equivalence is transferred to the rule creation process. This tool offers a significant advantage over CROW and MEWE by allowing transformations to any section of a Wasm program, being not limited to the code section. Moreover, it benefits from a virtually cost-free e-graph traversal process and avoids the *overlapping* issue seen in CROW and MEWE, as detailed in 3.1. In addition, since WASM-MUTATE operates at the binary level, it can modify functions incorporated by the WebAssembly producer itself. For example, this is the case of the *wasm32-wasi* architecture. While the original program might have a few lines of code, the underlying compiler might inject more functions to support the *wasm32-wasi* architecture. Thus, augmenting the diversification space available to WASM-MUTATE.

WASM-MUTATE outperforms CROW and MEWE capabilities in terms of number of generated variants. Yet, the changes made by WASM-

MUTATE might not be as preserved as the ones generated by CROW and MEWE. Thus, the variants generated by WASM-MUTATE might be more susceptible to be reversed, e.g. by further optimization passes.

3.4.3 Security guarantees

CROW and MEWE generate distinct and highly preserved code variants. This means that these variants, each with unique WebAssembly codes, maintain their distinctiveness even after JIT compilers translate them into machine codes. This distinctive feature significantly reduces the impact to side-channel attacks that exploit specific machine code instructions, e.g., port contention [?]. WASM-MUTATE, while offering slightly reduced preservation in its generated variants compared to CROW and MEWE, still maintains the same security guarantees excepting the multivariant cases. Its ability to produce a greater number of variants can offset this preservation shortfall.

Furthermore, CROW and MEWE enhance security against timing-based attacks by creating variants that exhibit a wide range of execution times. This strategy is especially prominent in MEWE's approach, which develops multivariants functioning on randomizing execution paths, thereby thwarting attempts at timing-based inference attacks. Consequently, attackers might find it exceedingly difficult to identify a specific variant through time profiling of a MEWE multivariant, ensuring a heightened level of security (See section ??). Adding another layer to this security framework, the integration of diverse variants into multivariants can potentially disrupt the functioning of dynamic analysis tools such as symbolic executors [?]. For example, different control flows through a random discriminator, which exponentially increases the number of paths, rendering them virtually unexplorable.

An advantege of WASM-MUTATE, compared to CROW and MEWE, is its capacity to transform non-code sections without impacting the runtime behavior of the original variant, a strategy that effectively shields against static binary analysis, including malware detection based on signature sets [?]. For instance, it can modify the type section of a WebAssembly program, a section typically utilized only for function signature validation during compilation and validation processes by the host engine. This thwarts compiler identification techniques, such as fingerprinting. Besides, it can be used for masquerading as a different compilation source. Thus, reducing the fingerprinting surface available to attackers.

CROW, MEWE, and WASM-MUTATE all have the capability to alter the original program structure, either by eliminating dead code or by introducing additional elements. From a static perspective, such alterations serve to reduce potential attack surfaces, thereby impeding signature-based identification techniques as noted in [?]. Modifying the layout of a WebAssembly program invariably affects its unmanaged memory during runtime, a segment not overseen by the WebAssembly program itself (see section ?? for a detailed discussion on unmanaged memory). This aspect is especially important for CROW and MEWE, given that they do not directly address the WebAssembly memory model, i.e., they do not directly diversify memory accesses. For example, the *constant inferring* transformations, which significantly alter the layout of program variants, affects unmanaged memory elements such as the returning address of a function. On the other hand, WASM-MUTATE not only affects unmanaged memory through changes in the WebAssembly program layout. It also adds rewriting rules to transform managed memory instructions. Memory alterations, either to the unmanaged or managed memories, have substantial implications for security. For instance, they can counteract attacks by eliminating potential jump points that facilitate malicious activities within the binary, a preventive measure highlighted by Narayan et al. [?].

3.5 Accompanying artifacts

This thesis is accompanied by the source code of the three contributions, CROW, MEWE and WASM-MUTATE. The source code is accessible through the links:

1. CROW: `https://github.com/KTH/slumps`
2. MEWE: `https://github.com/Jacarte/MEWE`
3. WASM-MUTATE: **TODO** URL

Our software artifacts are licensed under the MIT License. The dependent source codes, such as LLVM, are licensed under their original conditions.

3.6 Conclusions

In this chapter we discuss the technical specifics underlying our primary technical contributions. We elucidate the mechanisms through which CROW engenders program variants, all in the service of advancing software diversification. Following this, we outline the conceptual framework for a universal LLVM superdiversifier, laying a foundation for broader applicability and versatility. Subsequently, we turn our attention to MEWE, offering a detailed examination of its role in forging a robust MVE system. We also explore the inner workings of WASM-MUTATE, highlighting its pioneering utilization of an e-graph traversal algorithm to spawn Wasm program variants. Remarkably, we undertake a comparative analysis of the three tools, delineating their respective benefits and limitations, alongside the potential security assurances they provide upon the program variants derived from them.

In **??**, we present four use cases that support the exploitation of these tools. **??** serves to bridge theory with practice, showcasing the tangible impacts and benefits realized through the deployment of CROW, MEWE, and WASM-MUTATE.

04 EXPLOITING SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

4.1 Offensive Software Diversification

4.1.1 **Use case 1:** Automatic testing and fuzzing of WebAssembly consumers

TODO We explain the CVE. Make the explanation around "indirect memory diversification"

4.1.2 **Use case 2:** WebAssembly malware evasion

TODO The malware evasion paper

4.2 Defensive Software Diversification

4.2.1 **Use case 3:** Multivariant execution at the Edge

TODO Disturbing of execution time. Go around the web timing attacks.

4.2.2 **Use case 4:** Speculative Side-channel protection

In concrete, distributing the unmodified binary to 100 machines would, essentially, creates 100 homogeneously vulnerable machines. However, let us illustrate the case with a different approach: each time the binary is replicated onto a different machine, we distribute a unique variant instead of the original binary. If we disseminate a unique variant, with X stacked transformations, to each machine, every system would run a distinct Wasm binary. Based on our findings, even when some binaries are still vulnerable, we can confidently say that if 100 variants of a vulnerable program, each furnished with X stacked transformations, are distributed, the impact of any potential attack is considerably mitigated. While it's true that some variants may retain their original vulnerabilities, not all of them do. This significantly enhances overall security. Further reinforcing this point, let's consider the case of btb_leakage. In this scenario, a suite of 100 variants, each featuring at least 200 stacked transformations, ensures full protection against potential threats, effectively securing the entire infrastructure. Moreover, considering the results for the ret2spec attack, this property holds for the whole population of generated variants, despite the number of stacked transformations. Therefore, WASM-MUTATE as a software diversification tool, is a peemptive solution to potential attacks.

TODO Go around the last paper

05 CONCLUSIONS AND FUTURE WORK